CSA Coursework:
Game of Life

George Edward Nechitoaia (dp19681)
Vlad Andrei Bucur (ot19588)

Stage 1 - Parallel Implementation

Our goal is to deliver a parallel implementation of Game of Life **using multiple worker goroutines** on a single machine.

The initialisation of the project consisted of creating new channels in **gol.go**, to create a connection between IO (**io.go**) and the Distribuitor (**distributor.go**). This connection made the distributor able to read a PGM file, parse it in computable variables (parameters + board) and save the current state of the board to another PGM file whenever it is required.

The main starting **goroutines** are the **distributor** and the **IO**. They start in parallel making the game able to input/output data at any moment, while computing next stages of the board.

Our first running version was a **single threaded** Game of Life. It processed the logic for every next state imperatively. The events for CellFlipped, TurnComplete and FinalTurnComplete are updated while the board is computing, making the SDL able to visualise the stages of the board.

For **improving** the time complexity, we splitted our computations between the main manager (distributor function) and the workers (worker function).

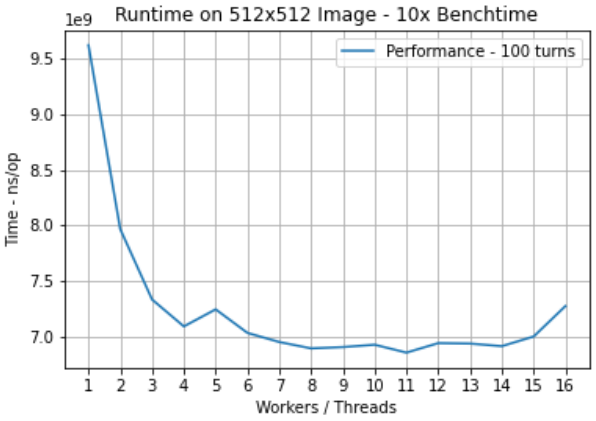
Every **worker** is a **goroutine** started in parallel and computes a different area of the board in order to **increase efficiency**. The board is splitted in rectangles of equal heights, each one with the same width as the given input. In case the board cannot be splitted equally, we leave the last worker to access a larger area, to make sure we have all the board computed. In case of edge computing, a specific worker is able only to look at the whole board to get the number of neighbours. This way, we **avoid any data race**.

The distribuitor uses a boolean channel to wait for every worker to finish, and then merges the computed pieces of the board.

We use the **select statement** to wait on multiple channels. This way, a **ticker channel**, which keeps track of time, reports every 2 seconds the alive cells. Furthermore we listen within the same select statement for **keyPressed channel**. We use this channel to report any received control keys from the keyboard. This way we can interact with the SDL visualisation, making the distribuitor able to save(‘s’ key),

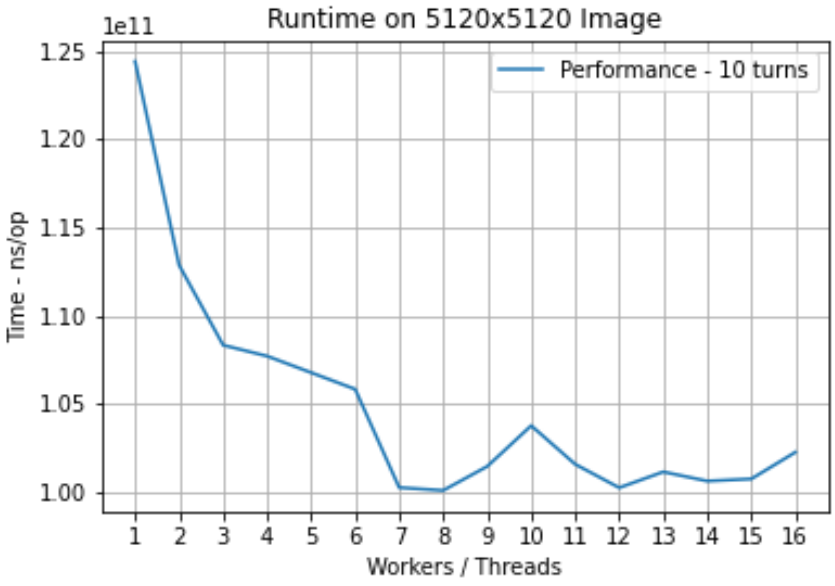
pause(‘p’ key) and terminate(‘q’ key) the program. In case of pressing the ‘p’ key, the distribuitor goes into an infinite loop, pausing the process. If the ‘p’ key is pressed again, it will break the loop, continuing the process.

We ran **benchmarks** to analyse and interpret the progress of our implementations, as it is run **with 1 or more threads**. The graph on the right gives a visualisation of the benchmarks run on the **512x512 image**, for **100 turns**. To make the analysis as reliable as possible we ran each benchmark ten times per worker, and took **the average**. The trend-line is falling as more workers are used in the game, proving that it is increasing efficiency by using less time per operation. However there are some small fluctuations where the time of the execution seems to increase, but it is recognised as a limitation of the CPU. On more powerful machines, running on a bigger image for more than 100 turns the efficiency of the algorithm would be more obvious than on the example provided here.



To summarise, our implementation scales better in terms of time complexity when more workers are added. This way the board is splitted in more processes which are done simultaneously.

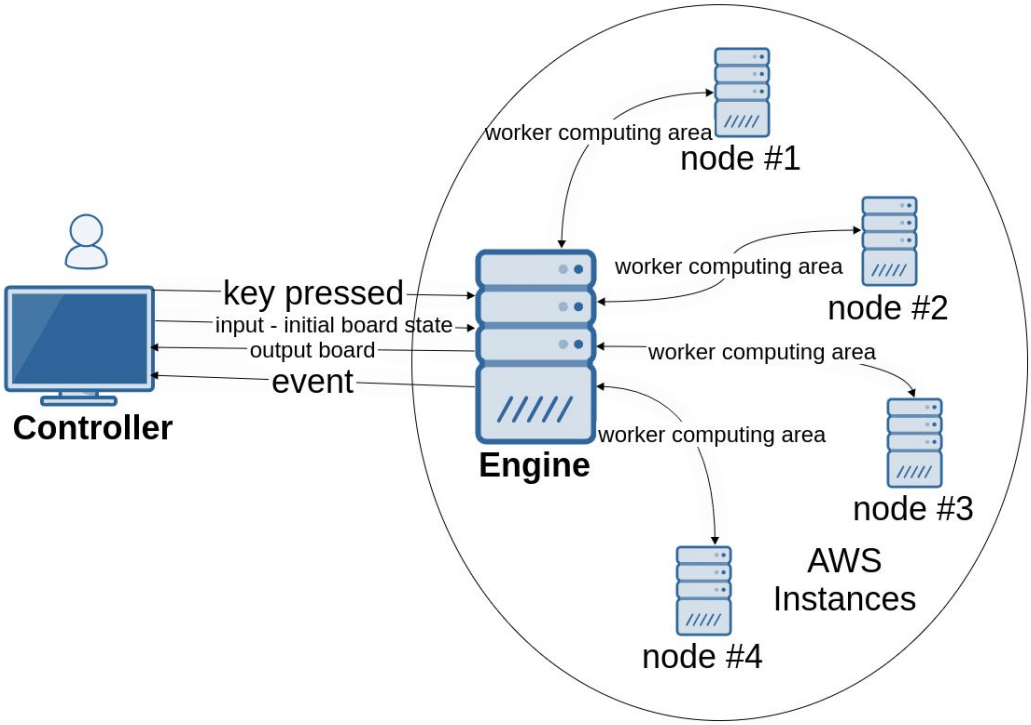
We used benchmarks to test and debug the implementation and tried to optimize it. We encountered a major throwback when the implementation was completed, it passed all the tests but the single threaded runtime was giving better results than the multithreaded one. We found the bug and solved it: the program was starting a goroutine and the execution would not continue until it was finished. This bug transformed the parallel implementation into a sequential one, which was incorrect.



Stage 2 - Distributed Implementation

Our goal is to deliver a distributed implementation of Game of Life that uses a number of AWS nodes to cooperatively calculate the new state of the board, and communicate state between machines over a network.

Our implementation is made up of the following components: a local Controller, an Engine and 4 Worker Nodes. For achieving the desired behavior we made a Distributed Client Server Model. At high level, the engine plays the role of the server and the others play the role of the clients.



Clients:

The **controller** is designed to provide an input, an initial state to the server. There are several functionalities if you are pressing the keys (eg. you can pause the game).

The **nodes** are similar to the workers from the concurrent implementation. They are working in parallel to compute the whole board. Data is received from the engine when a new turn starts and sends back the result after the node computes the logic of Game of Life. We used the halo exchange scheme, in which the nodes are communicating only by edges, to increase the efficiency of data transfer between the

nodes and the server. Furthermore, we made each node to include **multithreading**, making the system more efficient in terms of time complexity.

Server:

The **engine** is the connection between the controller and the nodes. It is also **the manager of the nodes**. The main functionality is to split the work provided by the controller and send it to the nodes to be computed.

The server **listens** to the TCP 8080 port while the clients are **dialing** the IP:port of the server. We have used client IDs to identify the nodes and the controller connections. The communication is made by message passing through strings. These strings have integrated codes in order to filter and select the incoming data.

About the encoding of the messages (strings):

- The **‘<n> map’** code is the main data transfer, as the goal of the game is the continuously computing board. **<n>** is represented by the client id. It encodes the board, threads number, height, width, turns and client identification parameters into a single string. The board is encoded in a different format. It consists of 1 and 0 characters, where 1 represents an alive cell and 0 represents a dead cell.

  Example: *2 map 3 3 1 4 101 111 100* -> the information is from the **client ID#2,** it has **3 width, 3 height, current turn is 1, 4 threads** and the **world[][]={(1,0,1),(1,1,1),(1,0,0)}** - where 0 means dead cell and 1 means alive cell.

- The **‘cf’** code is for encoding the CellFlipped events. It encodes the numerical values of the following variables: x, y and turn.

  Example: *cf 1 1 2* -> **CellFlipped event for the cell with coordinates (1,1) at turn = 2**

- The **‘tc’** code is for encoding the TurnComplete events. It encodes the numerical value of the turn variable.

  Example: *tc 5* -> **TurnComplete event for turn = 5**

- The **‘ftc’** code is for encoding the FinalTurnComplete events. It encodes the numerical values of the following variables: turn and chunks of numeric data (x, y) for all the alive cells after completing the last turn.

Example: *ftc 2 x:1;y:2 x:3;y:1* -> **FinalTurnComplete event at turn = 2 with alive cells from the coordinates (1,2) and (3,1)**

- The '**acc'** code is for encoding the AliveCellsCount events. It encodes the numerical value of the variable turn and the number of the alive cells at the specific turn.
  Example: *acc 3 5* -> **AliveCellsCount event at turn = 3 with 5 cells alive**

- The '**key'** code is for encoding a key pressed:
  1. 'keypauseTheGame' - when 'p' is pressed - sends a request to pause the game or continue the game if it was already paused
  2. 'keysaveTheGame' - when 's' is pressed - sends a request to save the current board
  3. 'keyquitTheGame' - when 'q' is pressed - close the connection between the controller and the engine, leaving the engine to continue the processing
  4. 'keyshutDown' - when 'k' is pressed - request to completely shut down every component of the distributed system

All the **events encodings** are taking part in the engine and the decodings are taking part in the controller. The **key encodings** and filtering are taking part within all 3 components in order to control the runtime of the processes. This way the **SDL Visualisation** works properly, being **responsive** to the user control inputs (if any keys are pressed). To make sure we have a **DRY code** we made different functions to encode and decode our messages between the distributed system.

It is necessary to start the server first, then the nodes to make sure there is an established connection between the processing components. The system will start working after the controller connects and sends an initial state of the board to the engine. If any component of our system crashes it will stop producing any output and the whole distributed system has to be restarted.

Our distributed system is efficient because it computes data using multithreading on 4 different machines. It can be **improved** with:

- connecting more than 4 AWS nodes to the engine (server)
- in case one of the nodes crashes adapt to the situation and continue computing the Game of Life